Vinícius Fratin Netto

Análise da Criticidade de Falhas Transientes em Processadores Paralelos para Enrobustecimento Seletivo

Brasil

2019/2

Vinícius Fratin Netto

Análise da Criticidade de Falhas Transientes em Processadores Paralelos para Enrobustecimento Seletivo

Trabalho de Graduação. Universidade Federal do Rio Grande do Sul. Tolerância a falhas.

Universidade Federal do Rio Grande do Sul

Instituto de Informática

Bacharelado em Engenharia de Computação

Orientador: Paolo Rech

Coorientador: Daniel Oliveira

Brasil

2019/2

Vinícius Fratin Netto
Análise da Criticidade de Falhas Transientes em Processadores Paralelos para Enrobustecimento Seletivo/ Vinícius Fratin Netto. – Brasil, 2019/2-
46p. : 8 il. ; 30 cm.

Orientador: Paolo Rech

Trabalho de Graduação II – Universidade Federal do Rio Grande do Sul
Instituto de Informática
Bacharelado em Engenharia de Computação, 2019/2.

1. Trabalho. 2. Conclusão. 3. Curso. I. Paolo Rech. II. Universidade Federal do Rio Grande do Sul (UFRGS). III. Instituto de Informática. IV. Análise da Criticidade de Falhas Transientes em Processadores Paralelos para Enrobustecimento Seletivo.

Resumo

Este documento tem como objetivo descrever o trabalho realizado ao longo do período de um ano, dedicado ao trabalho de graduação.

O trabalho consiste em realizar uma análise de criticidade dos erros na saída de aplicações paralelas que sofrem falhas durante sua execução. O modelo de falhas considerado neste trabalho é o modelo de falhas causadas devido à radiação cósmica. Como resultado dessa análise, é possível obter uma classificação de quais variáveis são candidatas para a técnica de enrobustecimento seletivo.

Adicionalmente, a implementação de uma ferramenta que provê a infraestrutura para a aplicação automática de enrobustecimento seletivo foi realizada.

Palavras-chave: trabalho de graduação, radiação, falhas, criticidade, enrobustecimento seletivo.

Abstract

This document aims to describe the work made during the one year time period dedicated to the undergraduate thesis.

The work consists of providing an analysis of the criticality of the errors in the output of parallel application that experience faults during their execution. The considered fault model is that of faults caused by cosmic radiation. As a result of this analysis, it is shown that it is possible to obtain a classification of which variables are candidates for the selective hardening technique.

Additionaly, the implementation of a tool that provides the required infrastructure for automatic application of the selective hardening was achieved.

Keywords: undergraduate thesis, radiation, faults, criticality, selective hardening.

Lista de ilustrações

Lista de quadros

Sumário

1 Introdução

Este documento descreve as tarefas realizadas ao longo do trabalho de graduação. Assim, são descritas as propostas do trabalho realizado, o embasamento teórico necessário, a análise da criticidade das falhas injetadas nas aplicações e, finalmente, a implementação de uma ferramenta que possui a infraestrutura necessária para a aplicação da técnica de enrobustecimento seletivo.

A proposta do trabalho é uma análise específica da criticidade dos erros na saída de aplicações que sofrem alguma falha durante sua execução. A criticidade de um erro é definida por quanto distante é um valor na saída de uma aplicação em relação ao valor esperado. Essa análise será utilizada como base para a aplicação em uma outra técnica de tolerância a falhas em nível de código-fonte, denominada *enrobustecimento seletivo*, que pode fornecer uma detecção de falhas transientes apreciável. No contexto deste trabalho, a qualidade de mecanismo de detecção de falhas é medida a partir do tempo adicional necessário para a detecção e de quais circunstâncias as falhas podem ser efetivamente detectadas. Considere, por exemplo, uma aplicação que recebe uma dada entrada, gera uma saída dependente dessa entrada, e que é executada em uma plataforma com somente um processador. Uma forma simples de detecção, porém onerosa em termos de performance, consiste em executar a aplicação duas vezes para a mesma entrada. Caso a saída gerada seja a mesma para ambas as execuções, a probabilidade de que uma falha transiente tenha ocorrido nos dois casos e provocado exatamente o mesmo resultado é baixa. Caso a saída seja diferente entre as execuções, é possível constatar que houve uma falha em alguma delas (apesar de não ser possível afirmar em qual das execuções a falha ocorreu). Esse método, que é uma possível implementação da abordagem conhecida como *Redundância Modular Dupla* (DMR, do inglês *Dual Modular Redundancy*), provê um mecanismo de detecção de falhas que detecta falhas em diversos casos em que acontecem. Porém, o tempo adicional para essa detecção é o dobro do tempo de execução da aplicação, uma vez que são necessárias duas execuções sequenciais.

É importante notar que a eficácia do DMR depende de como é realizada a implementação da técnica. Por exemplo, duas execuções em sequência provavelmente detectam mais falhas do que uma dupla chamada de função dentro de um código. Isso é possível pois dentre os diversos componentes que podem sofrer alguma falha está, por exemplo, a memória cache. Se um dispositivo com múltiplos núcleos mantém uma memória cache compartilhada e uma falha corrompe um certo dado da cache, todos os núcleos que utilizam esse dado podem produzir o mesmo resultado incorreto, que não será detectado se a comparação for realizada entre núcleos. Como a análise das falhas neste trabalho é realizada apenas no nível de software, esse tipo de cenário não é considerado, pois a

memória cache é transparente para as aplicações.

A técnica de enrobustecimento seletivo consiste em selecionar os componentes do código-fonte da aplicação que mais afetam a saída na presença de uma falha e, então, realizar uma abordagem DMR apenas nesses componentes. Com isso, teoricamente, é possível aumentar a eficácia da detecção de falhas significativamente de uma forma que não haja impactos significativos de performance. A análise conduzida neste trabalho provê uma classificação possível das variáveis de uma aplicação que pode ser utilizada para realizar o enrobustecimento seletivo. Essa análise leva em consideração a criticidade dos dados incorretos na saída. Por exemplo, certas variáveis podem levar a erros muito mais acentuados na saída do que outras variáveis na presença de um bit-flip. Nesse caso, convém considerar a possibilidade de duplicar as variáveis mais críticas apenas, uma vez que isso tanto restringe a quantidade total de variáveis candidatas para a duplicação quanto atinge apenas as variáveis que realmente possuem impactos significativos na saída e, portanto, pode levar a resultados mais satisfatórios.

O tipo de falhas considerado neste trabalho são falhas provocadas devido a radiação ionizante. A principal fonte de radiação ionizante que provoca falhas em dispositivos eletrônicos modernos são raios cósmicos que atravessam a atmosfera terrestre e são capazes de chegar ao nível do mar. Entre as partículas capazes de provocar ionização dos átomos presentes em dispositivos semicondutores estão os nêutrons (BAUMANN, 2005). Falhas devidas a radiação ionizante podem provocar desde um erro de execução de uma aplicação até uma perda total do dispositivo onde a falha ocorreu. A radiação ionizante é um problema importante para *HPC* (*Computação de Alta Performance*, do inglês *High Performance Computing*), uma vez que supercomputadores geralmente consistem em centenas ou milhares de dispositivos de processamento paralelo executando a mesma aplicação simultaneamente. A probabilidade da radiação ionizante provocar uma falha em um único dispositivo é baixa, mas é substancialmente aumentada devido à presença dos milhares de outros dispositivos interconectados. Além de HPC, aplicações críticas como aviões e satélites são bastante sensíveis a falhas provocadas por esse tipo de problema, pois operam em alturas elevadas e, portanto, mais expostas à radiação ionizante. Alguns modelos de falhas que simulam o comportamento das falhas causadas por radiação ionizante serão abordados neste trabalho.

Como uma última proposta para este trabalho, é realizada a implementação de uma ferramenta capaz de gerar todas as estruturas de dados necessárias para realizar diversas operações sobre um código-fonte qualquer escrito na linguagem C11. A ferramenta faz a análise de todos os caminhos do código que levam a operações de leitura e de escrita das variáveis. Assim, em trabalhos futuros, será possível utilizar essa ferramenta como uma plataforma para a realização de etapas desejáveis para a implementação do enrobustecimento seletivo. De fato, duas das operações envolvidas no enrobustecimento

seletivo são uma instrumentação dinâmica e a duplicação da leitura e da escrita das variáveis desejadas. A instrumentação dinâmica pode ser utilizada para obter o número total de leituras e de escritas em cada uma das variáveis do código-fonte considerado. Dessa forma, é possível selecionar as variáveis que possuem o menor overhead de duplicação. Finalmente, a duplicação de leitura e de escrita apenas das variáveis escolhidas é o que realiza a implementação efetiva do enrobustecimento seletivo.

A implementação da instrumentação dinâmica e da duplicação, porém, foge do escopo deste trabalho. Essas funcionalidades poderão implementadas no futuro através da infraestrutura provida pela ferramenta implementada.

Os detalhes sobre a proposta do trabalho estão detalhados no Capítulo 2. Os detalhes sobre a radiação ionizante, modelos de falhas e o injetor de falhas, bem como a justificativa para a análise conduzida neste trabalho estão contidos no Capítulo 3. O ambiente em que os testes foram conduzidos está descrito no Capítulo 4. A análise de criticidade e a discussão sobre os resultados obtidos estão no Capítulo 5. A descrição da ferramenta de automatização do enrobustecimento seletivo está contida no Capítulo 6. Finalmente, as considerações finais do trabalho estão no capítulo de Conclusão.

2 Proposta

O trabalho consiste, basicamente, de duas seções.

A primeira seção envolve a análise da criticidade das falhas durante a execução de um conjunto de aplicações para HPC. A partir dessa análise, será possível obter uma classificação das variáveis candidatas a receber o enrobustecimento seletivo. A análise será conduzida para três diferentes benchmarks relevantes em aplicações de HPC. Cada aplicação será executada em um ambiente de injeção de falhas por software que visa simular o modelo de falhas presente na radiação ionizante.

A segunda seção envolve o desenvolvimento de uma ferramenta que permite a aplicação sistemática da técnica de enrobustecimento seletivo em aplicações escritas na linguagem C11 (NETTO, 2019). Ainda, a ferramenta deve ser capaz de gerar códigos enrobustecidos para aplicações que utilizam construções do *OpenMP*, uma API para o desenvolvimento de aplicações paralelas. Porém, a ferramenta não possui limitação em relação a qual tipo de API para programação paralela será utilizada, pois a proposta não depende de recursos que apenas o OpenMP possui. A escolha do OpenMP é apenas uma decisão de projeto.

3 Embasamento

Neste capítulo, são detalhados os principais aspectos teóricos e práticos que sustentam a proposta deste trabalho. Na seção 3.1, são abordados alguns dos problemas que a radiação ionizante apresenta para dispositivos semicondutores comerciais. Na seção 3.2, é detalhado o procedimento de injeção de falhas para avaliação da taxa de falhas da aplicação. Na seção 3.3, são apresentados os trabalhos acadêmicos com resultados que indicam a viabilidade do enrobustecimento seletivo como técnica de detecção e mitigação de falhas devidas à radiação ionizante.

3.1 Radiação Ionizante

A Terra recebe uma grande quantidade de raios cósmicos vindos de diferentes pontos do espaço. Quando esses raios cósmicos colidem com a camada superior da atmosfera terrestre, uma cascata de partículas, composta principalmente por nêutrons, é produzida. Um fluxo de aproximadamente 13 $\mathrm{n}/(\mathrm{cm}^2 \times \mathrm{h})$ atinge a superfície, e o número de nêutrons aumenta exponencialmente com a altitude (JEDEC, 2006).

A interação de um dispositivo semicondutor com um nêutron pode provocar comportamentos inesperados nos estados dos transistores como, por exemplo, *single bit-flips* (i.e., quando apenas um bit de um certo dado é alterado) e *multiple bit-flips* (i.e., quando dois ou mais bits de um certo dado são alterados). Ainda, a interação pode provocar picos de corrente em circuitos lógicos que, na ocorrência de um *latch-up* (i.e., um tipo de curto-circuito que pode ocorrer em circuitos integrados), podem levar a uma falha permanente no circuito. Esse tipo de falha (i.e., falhas que podem levar a danos físicos do dispositivo) não será considerada neste trabalho.

Apesar de nêutrons não serem capazes de ionizar átomos da mesma forma que outras partículas carregadas podem (nêutrons não possuem carga elétrica), a colisão de um nêutron de energia suficientemente grande com o núcleo de um átomo pode resultar na produção de isótopos, radiação alfa, radiação beta e radiação gama. A radiação gama, por sua vez, pode possuir energia suficiente para expulsar um elétron do átomo original, originando um íon positivamente carregado. O processo de ionização pode provocar correntes transientes nos transistores dos dispositivos semicondutores que podem ser grandes o suficiente para alterar o estado desses transistores (BAUMANN, 2005). Um cenário possível, por exemplo, é o de um transistor na saída de uma porta lógica de um microprocessador que é atingido por um nêutron de alta energia. Se a radiação ionizante altera o estado do transistor (e.g., da região de corte para a região linear), a tensão de saída da porta lógica pode ser diferente da tensão esperada na operação correta. Em particular, a tensão de saída pode representar

um bit de valor 0 em vez de um bit de valor 1, ou vice-versa. Esse é um exemplo de *bit-flip*.

Devido à diminuição das dimensões dos transistores e à quantidade de recursos disponíveis, os dispositivos eletrônicos atuais são bastante suscetíveis a falhas devidas à radiação ionizante. De fato, partículas ionizantes possuem energia suficiente para corromper dados armazenados em memórias SRAM ou para afetar o resultado de operações lógicas (BAUMANN, 2005; SILVA et al., 2017; SHEIKH et al., 2017). É esperado que a tendência em relação à confiabilidade dos dispositivos eletrônicos se torne pior conforme o tempo passa. Com supercomputadores se aproximando do *exascale* (i.e., capazes de atingir pelo menos 1 hexaFLOPS, ou 10^{18} operações de ponto flutuante por segundo), é esperado que as dimensões dos transistores sejam reduzidas em até três vezes em relação ao tamanho atual. Assim, a carga armazenada em um transistor se tornará menor, aumentando a probabilidade de que a radiação induza uma falha (GEIST, 2016; MAHATME et al., 2011).

Em relação a sistemas de computação, a radiação pode corromper elementos de memória, como registradores, caches ou latches, mas também elementos lógicos, como portas lógicas e o escalonador. A interação com a radiação pode causar um dos seguintes cenários: (1) Nenhum efeito na saída da aplicação (a falha é mascarada ou o dado corrompido não é utilizado); (2) *Corrompimento Silencioso do Dado* (SDC, do inglês *Silent Data Corruption*), i.e., uma saída corrompida; (3) *Crash* da aplicação e (4) um *hang* do dispositivo, que deve ser reiniciado para recuperar a funcionalidade correta. Dentre esses cenários, o cenário (2) é preocupante, pois a interação com a radiação não é detectada e, portanto, a saída da computação pode ser corrompida inesperadamente. Os cenários (3) e (4) podem causar perdas de performance ou perda de dados caso nenhum ponto de recuperação tenha sido salvo. SDCs têm mais chance de serem causados por falhas na memória, nas portas lógicas ou no escalonador, apesar de que falhas no escalonador também podem causar hangs. Falhas nas caches de instruções, nos barramentos de comunicação ou na lógica de controle provavelmente causarão hangs ou crashes (RECH et al., 2014).

Aplicações científicas de larga escala usualmente possuem longos tempos de execução, variando de algumas horas até alguns dias, e envolvem um grande número de dispositivos (e.g., o supercomputador Titan, um dos mais rápidos supercomputadores atuais (TOP500, 2016), é composto por mais de 18000 GPUs). Devido à larga escala e ao longo tempo de execução, as aplicações científicas mais complexas podem ser interrompidas devido a crashes ou hangs, bem como SDCs na saída (GEIST, 2016). Como uma ilustração, o *Tempo Médio para uma Falha* (MTTF, do inglês *Mean Time to Failure*) é da ordem de algumas dezenas de horas (TIWARI et al., 2015). Um sistema *exascale* deve ser 55× mais rápido do que o Titan. Portanto, é impraticável construir um sistema exascale com a tecnologia utilizada no Titan, pois uma taxa de falhas 55× maior, bem como uma potência dissipada 55× maior, é inviável.

Failures In Time (FIT), que representa a quantidade de falhas a cada 10^9 horas, é

a métrica universal adotada para medir a confiabilidade de dispositivos e de aplicações. Por exemplo, de acordo com o ISO 26262, a taxa de falhas de sistema de direção autônomo deve ser limitada a 10 FIT, que é um valor extremamente pequeno quando consideramos a taxa média de falhas induzidas por nêutrons (OLIVEIRA et al., 2016). Porém, o FIT considera que todas as partes físicas do dispositivo são passíveis de uma interação com um nêutron, por exemplo. Portanto, é possível obter o FIT de uma aplicação através de um teste com canhão de nêutrons, por exemplo. Nesse cenário, todo o dispositivo é irradiado, e o FIT pode ser medido. Isso torna a utilização dessa métrica impossível para este trabalho, pois a injeção de falhas é utilizada apenas no nível da aplicação, tornando o hardware inacessível para as injeções. Dessa forma, a métrica utilizada para avaliar a taxa de falhas da aplicação é o *PVF* (do inglês, *Program Vulnerability Factor*), que mede a probabilidade de uma falha injetada em uma variável aleatória da aplicação causar uma saída incorreta (OLIVEIRA, 2017). Assim, é possível obter informações sobre a taxa de falhas da aplicação utilizando apenas injeção de falhas por software.

3.2 Injeção de falhas

A maneira pela qual o PVF da aplicação será avaliado é através de um injetor de falhas. O injetor de falhas escolhido é o *CAROL-FI* (OLIVEIRA et al., 2017). Esse injetor de falhas possui um modelo de falhas que simula um único *bit-flip* em uma variável aleatória durante a execução de uma aplicação. Esse é um dos modelos suportados, mas, a princípio, é possível obter resultados interessantes apenas considerando esse modelo específico de falhas.

O injetor de falhas é capaz de injetar falhas apenas em um intervalo de tempo definido pelo usuário. Isso permite que as falhas sejam injetadas apenas durante a execução do kernel das aplicações. Para especificar esse e outros parâmetros, o injetor de falhas utiliza um arquivo de configuração. Esse arquivo deve ser previamente escrito pelo usuário e deve conter os caminhos no disco para o executável da aplicação e também deve conter os parâmetros de execução dessas aplicações.

Para cada arquivo de configuração, o injetor ainda requer o número desejado de iterações a serem executadas. Com isso, é possível injetar uma quantidade grande o suficiente de falhas para que se tenham dados estatisticamente significativos. Os resultados de cada execução são armazenados em arquivos de log.

3.3 Enrobustecimento seletivo

A principal ideia por trás do enrobustecimento seletivo é permitir uma maior taxa de detecção ou mitigação de falhas sem introduzir um *overhead* significativo no tempo de

execução da aplicação.

Em trabalhos existentes do grupo de pesquisa que o autor participa, já existem algumas análises do comportamento de aplicações utilizando a técnica de enrobustecimento seletivo (OLIVEIRA, 2017). Essas análises sugerem que é possível obter uma melhora significativa na confiabilidade da aplicação ao realizar a duplicação de uma ou mais variáveis sem que haja um impacto significativo no tempo de execução. No contexto de HPC, é comum a utilização de *checkpoints* periódicos para que seja possível continuar a execução de uma aplicação após a ocorrência de uma falha sem que seja necessário reiniciar completamente essa aplicação. Portanto, caso o enrobustecimento seletivo detecte alguma falha, o sistema pode realizar um *checkpoint rollback*, voltando ao estado salvo no último *checkpoint*.

A principal contribuição deste trabalho para esse estudo é prover mais uma análise de como e quais variáveis devem ser enrobustecidas. No caso deste trabalho, essa análise leva em consideração não somente o PVF de cada variável, mas também a criticidade que falhas ocorridas nessa variável possuem na saída. Adicionalmente, o trabalho provê um protótipo de ferramenta que poderá ser utilizada no futuro para automatizar o processo de enrobustecimento das variáveis. Essa ferramenta pode ser utilizada para aplicar o enrobustecimento das variáveis de acordo com a classificação desenvolvida nos próximos capítulos deste trabalho.

4 Ambiente de teste

Neste capítulo, são descritas as características do ambiente em que os testes foram executados.

4.1 Processador

O processador utilizado foi o Intel Core i5-3337U. Esse processador possui apenas 2 núcleos físicos, mas suporta até 4 threads devido à presença da tecnologia *Hyper-Threading*. Cada núcleo possui uma frequência de clock de 1.8 GHz.

O processador possui 3 níveis de memória cache. A cache L1 possui 128 KB, a cache L2 possui 512 KB, e a cache L3 possui 3 MB (INTEL, 2019).

4.2 Benchmarks

Para os fins deste trabalho, foram escolhidos 3 benchmarks utilizados em aplicações para supercomputadores. As aplicações são:

- **DGEMM (Double-precision General Matrix Multiplication)**: esse benchmark implementa uma versão otimizada do algoritmo de multiplicação de matrizes clássico para processadores paralelos. As entradas utilizadas para esse código são duas matrizes de 1024 linhas por 1024 colunas. A saída do algoritmo é a multiplicação entre as duas matrizes de entrada e, portanto, também é uma matriz de 1024 linhas por 1024 colunas.
- **Hotspot**: esse benchmark implementa uma versão discretizada da equação do calor para simular a distribuição de temperatura em um chip sujeito a uma certa distribuição de potência e de temperatura iniciais. As entradas são as distribuições de temperatura e de potência iniciais e o número de iterações desejado. A saída é a distribuição de temperatura após o período de tempo correspondente ao número de iterações inserido.
- **LavaMD**: esse benchmark simula a interação gravitacional entre um número grande de partículas em um espaço tridimensional. A simulação é feita através da divisão do espaço em caixas, que são especificadas como entrada. A saída é a posição e a velocidade de cada partícula.

Todos os benchmarks citados foram implementados utilizando OpenMP. O número de threads escolhido para as execuções foi de 4 threads. As entradas de cada benchmark

foram escolhidas de forma que o tempo de execução se situa entre 5 s e 10 s. Isso permite uma janela de tempo adequada para a injeção de falhas durante a execução do kernel (i.e., o código paralelo de fato) de cada aplicação.

Finalmente, todos os benchmarks foram compilados utilizando a flag -O0 do gcc. Isso foi feito com o intuito de não afetar a avaliação dos benchmarks devido a particularidades das otimizações do compilador.

4.3 Injeção de falhas

Como mencionado na seção 3.2, o injetor de falhas utilizado foi o CAROL-FI.

Os parâmetros do injetor foram configurados de forma que a injeção de uma falha é feita somente no intervalo de tempo correspondente à execução do kernel de cada aplicação.

Para cada aplicação foram injetadas, no mínimo, 4000 falhas. Desse número, em torno de 1000 SDCs foram detectados no total, resultando em aproximadamente 300 SDCs por cada aplicação. Essa quantidade permite a obtenção de uma amostra estatisticamente significativa para a análise dos resultados.

Cada injeção de falha demora um torno de 12 s. As injeções foram conduzidas em um tempo de, aproximadamente, uma semana, com o processador utilizado ininterruptamente.

5 Análise de Criticidade

Neste capítulo, é realizada a análise da criticidade das falhas injetadas com ênfase em quais variáveis foram afetadas e quais foram os efeitos nos valores dessas variáveis.

5.1 Motivação

A principal motivação para essa análise reside no fato de que o enrobustecimento seletivo pode ser muito mais eficiente do que uma duplicação total da execução de uma aplicação. De fato, teoricamente, é possível atingir uma diminuição de até 95% do PVF total de algumas aplicações com apenas um aumento de pouco mais de 6% no tempo de execução da aplicação (Oliveira; Navaux; Rech, 2019).

Com a análise da criticidade, é possível restringir ainda mais o conjunto de variáveis a serem consideradas para o enrobustecimento seletivo, uma vez que o relaxamento da definição da corretude dos dados da saída, em muitos casos, diminui o PVF total da aplicação.

5.2 Critério para análise de criticidade

O parâmetro utilizado para classificar qual falha é mais crítica entre um conjunto de mais de uma falha é o *erro relativo* entre o valor esperado na saída e valor efetivamente produzido na execução da aplicação. Todos os valores considerados são de ponto-flutuante. Dado o valor esperado, v_e, e o valor (possivelmente incorreto) de saída, v_s, o erro relativo, E_r, medido em porcentagem, é definido como

$$E_r = \left| \frac{v_e - v_s}{v_e} \right| \times 100\%. \tag{5.1}$$

Portanto, um erro relativo de 10% significa que o valor efetivamente produzido na saída da aplicação difere do valor esperado por ±10% deste valor. Logo, quanto menor o erro relativo, mais correta é a saída.

Para não ser necessário armazenar o erro relativo de cada valor de saída, o critério utilizado neste trabalho para avaliar a corretude de um conjunto de valores de saída é utilizar o *maior erro relativo* entre todos os valores de saída. Assim, um erro relativo máximo de 10% significa que todos os valores de saída diferem dos respectivos valores esperados por, no máximo, ±10% destes valores.

A análise aqui apresentada não considera o overhead resultante do enrobustecimento seletivo das variáveis estudadas, mas apenas a criticidade em relação aos erros nos valores

da saída. Uma análise mais detalhada sobre essa questão pode ser encontrada em (Oliveira; Navaux; Rech, 2019).

5.3 Cálculo do PVF

Para cada variável, o PVF é calculado como a razão entre o número de falhas que resultou em um SDC e o número total de falhas injetadas nessa variável. Dado o número de falhas que resultaram em um SDC, N_s, e o número total de falhas injetadas na variável, N, o PVF, medido em porcentagem, é calculado como

$$\text{PVF} = \frac{N_s}{N} \times 100\%. \tag{5.2}$$

Assim, se para uma certa variável o número de falhas que causaram SDCs foi 28, enquanto o número total de falhas injetadas foi de 67, então o PVF é de 41.79%.

O PVF também é uma medida interessante por permitir a comparação entre diferentes variáveis que, a princípio, possuem um número de injeções total diferente. Dessa forma, o PVF normaliza o resultado e permite que variáveis com números de injeções diferentes possam ser comparadas.

5.4 Resultados e Discussão

Para cada benchmark, foram selecionadas 6 variáveis distintas baseadas na quantidade total de injeções em cada variável. Isso foi feito para permitir a obtenção de dados mais estatisticamente significativos.

5.4.1 Geral

No contexto das aplicações como um todo, os resultados referentes ao PVF de cada aplicação podem ser observados no Quadro 1.

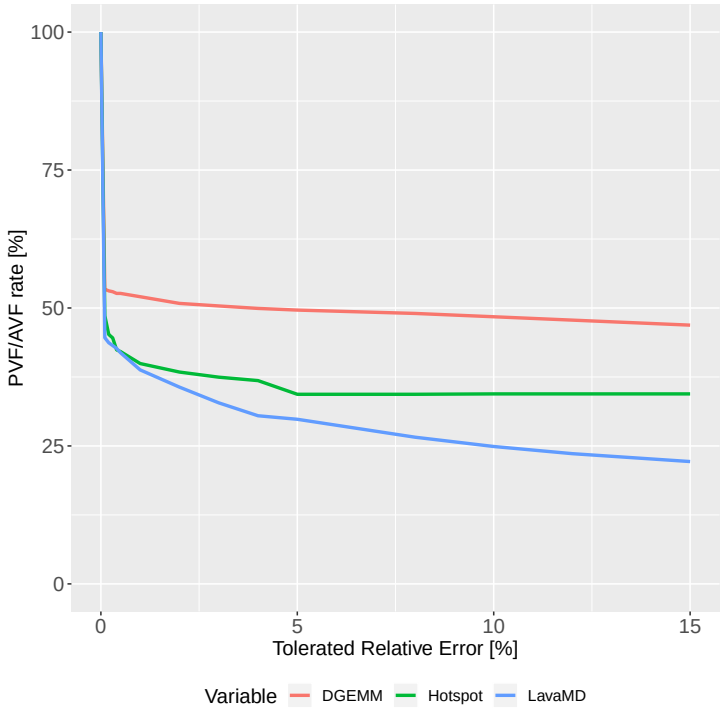
Por esses dados, notamos que o maior PVF ocorre para o DGEMM, enquanto o menor PVF ocorre para o Hotspot. Isso, porém, não é um indicativo do que acontece em relação à criticidade de cada variável, i.e., um maior PVF não significa uma maior criticidade.

Quadro 1 – PVF por aplicação.

Aplicação	**PVF** [%]
DGEMM	33.36
Hotspot	16.58
LavaMD	25.58

Fonte: Autor.

Figura 1 – PVF por erro relativo tolerado para todas as aplicações.



Fonte: Autor

De fato, isso pode ser parcialmente observado através da Figura 1. O eixo horizontal representa o erro relativo máximo tolerado na saída da aplicação. Mais especificamente, o erro relativo tolerado representa o maior erro relativo para o qual os valores da saída ainda são considerados corretos. Um erro relativo máximo tolerado de 5%, por exemplo, quer dizer que todos os valores na saída que possuem um erro relativo menor do que 5% são considerados como corretos ou, ao menos, como toleráveis. O eixo vertical representa a parcela do PVF de cada aplicação em função do erro relativo máximo tolerado na saída.

Pelo gráfico, é possível notar que todos os códigos, em menor ou maior escala, têm o PVF diminuído conforme o erro relativo máximo tolerado na saída aumenta. Porém, também é possível notar que a menor diminuição acontece para o DGEMM, enquanto a maior diminuição acontece para o LavaMD. Isso mostra que, mesmo que o PVF do Hotspot seja menor do que o PVF do LavaMD, o Hotspot perde para o LavaMD quando a criticidade dos valores da saída é levada em consideração.

Esses comportamentos serão ilustrados mais detalhadamente nas subseções que seguem.

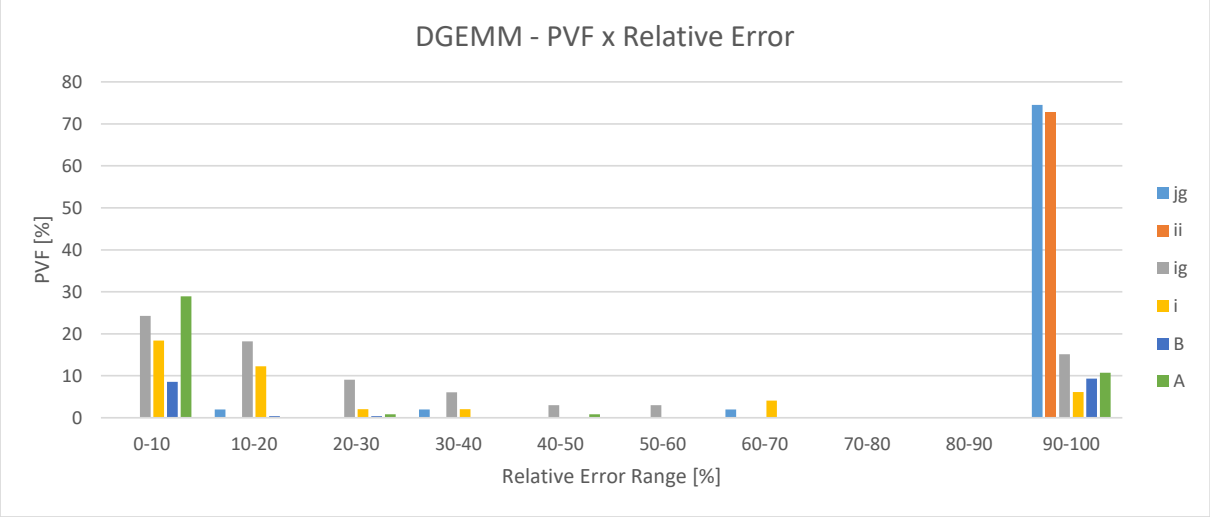
5.4.2 DGEMM

Na Figura 2, é possível verificar a distribuição do PVF de cada variável selecionada do DGEMM em função do erro relativo máximo na saída. No eixo horizontal, estão as faixas de erros relativos máximos. Na última faixa (i.e., de 90% a 100%), estão incluídas todas as falhas que produziram erros relativos máximos de 90% a 100% e, adicionalmente, todas as falhas que produziram erros relativos máximos maiores do que 100%. No eixo vertical, está o PVF da variável apenas considerando a faixa de erro relativo máximo respectiva.

No gráfico, é possível notar que a distribuição do PVF é distinta para cada variável e que, de fato, existem variáveis que possuem um impacto significativo na saída mesmo que resultados com uma certa faixa de erros relativos sejam considerados corretos. Considere, por exemplo, a variável A. Pelo gráfico, podemos notar que se uma faixa de 10% de erro relativo ainda é considerada correta, então uma parcela significativa do PVF é descartada (em torno de 30%). O mesmo não acontece com a variável jg, onde mesmo que uma faixa de 10% de erro relativo seja considerada correta o PVF ainda se mantém exatamente o mesmo. Mais ainda, é possível notar que esta variável é a que possui o maior PVF dentre todas as faixas de erro relativo e, pior ainda, esta é a faixa que corresponde a erros relativos máximos maiores do que ou iguais a 90%. Um comportamento semelhante a esse pode ser observado em relação à variável ii.

O padrão observado no gráfico sugere um ordenação possível para as variáveis que receberão o enrobustecimento seletivo. Neste caso, se o enrobustecimento seletivo for utilizado nas variáveis com o maior PVF na maior faixa de erros relativos, então o PVF

Figura 2 – PVF por erro relativo para o DGEMM.



Fonte: Autor

da aplicação como um todo pode melhorar significativamente. Dessa forma, a ordenação pode ser realizada como mostrado no Quadro 2.

Quadro 2 – Ordenação das variáveis do DGEMM para enrobustecimento seletivo.

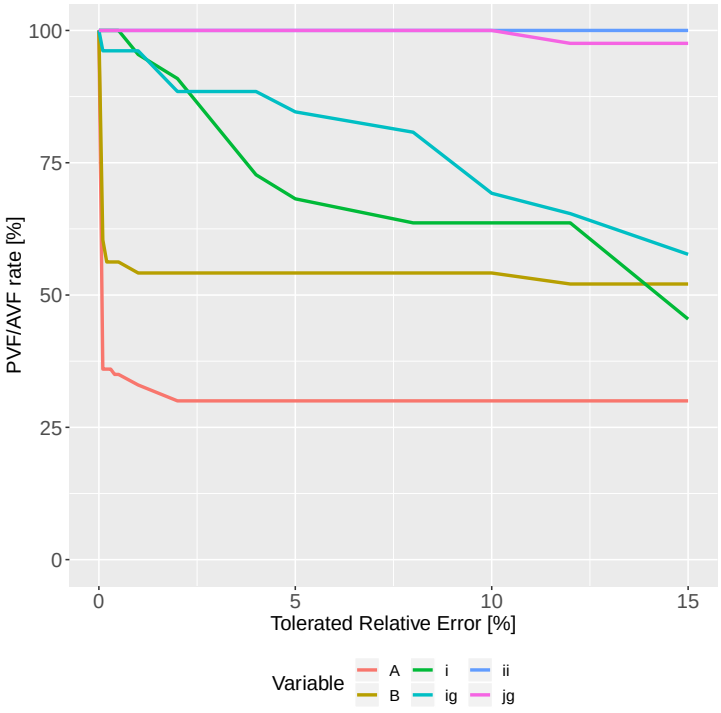
Classificação	**Variável**	**PVF** [%]
1ª	jg	2.06
2ª	ii	1.61
3ª	ig	1.31
4ª	A	5.04
5ª	B	1.76
6ª	i	1.11

Fonte: Autor.

Finalmente, na Figura 3, é possível observar o comportamento do PVF de cada variável em função do erro relativo máximo tolerado na saída. No eixo horizontal, está o erro relativo com que a aplicação ainda considera os dados da saída como corretos. No eixo vertical, está a porcentagem do PVF da variável que é diminuído em função do erro relativo máximo respectivo.

Como é possível observar pelo gráfico, o comportamento é distinto para cada uma das variáveis consideradas. Considere as variáveis *ii* e *jg*. Mesmo permitindo que o

Figura 3 – PVF por erro relativo tolerado para o DGEMM.



Fonte: Autor

erro relativo tolerado seja tão grande quanto 15%, o impacto no PVF dessas variáveis é ínfimo. Em contrapartida, para as variáveis A e B, a diminuição do PVF para um erro relativo tolerado de pouco mais de 0% já é significativa. Nesse caso, para a variável B, o PVF se torna aproximadamente 56.25%, enquanto para a variável A, o PVF se torna aproximadamente 37%.

5.4.3 Hotspot

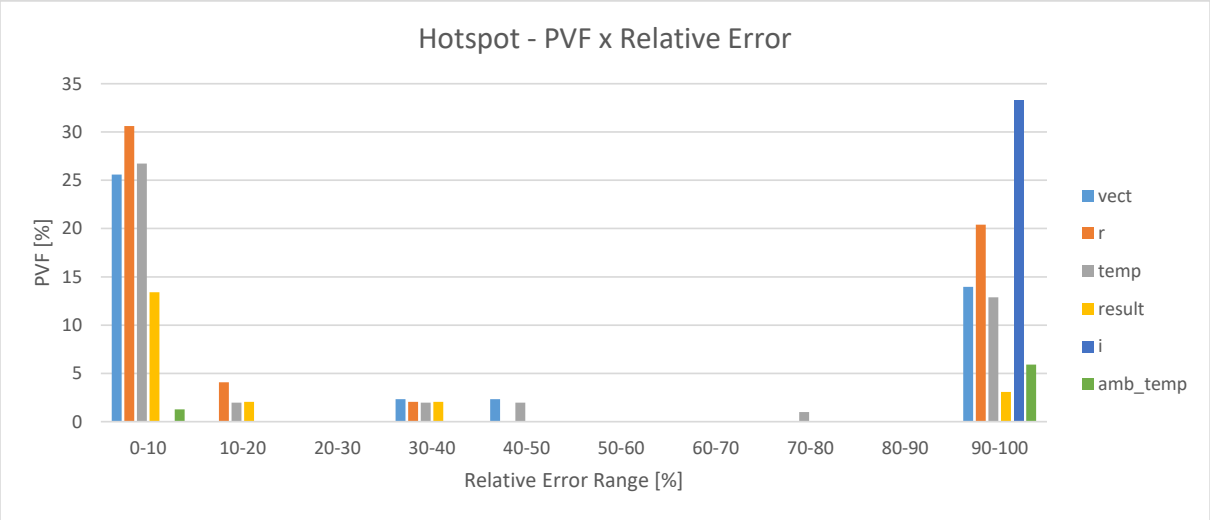
Na Figura 4, é possível observar a parcela do PVF de cada variável em função do erro relativo máximo na saída. Os eixos horizontal e vertical são exatamente como detalhado na subseção 5.4.2.

No gráfico, é possível notar que a distribuição do PVF é semelhante nas faixas de 0% até 10% e de mais do que 90%. Nesse caso, é possível observar que todas as variáveis, mesmo que estejam na faixa de 0% até 10%, possuem um PVF relativamente elevado na faixa de mais do que 90%. Isso nos permite concluir que, mesmo relaxando a noção de corretude para os valores dessas variáveis na saída, possivelmente a quantidade de falhas com erros relativos inaceitável ainda será grande.

Pelo gráfico, é possível realizar a ordenação das variáveis candidatas para o enrobustecimento seletivo. Essa ordenação pode ser verificada no Quadro 3.

Finalmente, na Figura 4, é possível verificar o comportamento do PVF de cada variável em função do erro relativo máximo permitido na saída. Os eixos horizontal e vertical são como descritos na subseção 5.4.2.

Figura 4 – PVF por erro relativo para o Hotspot.



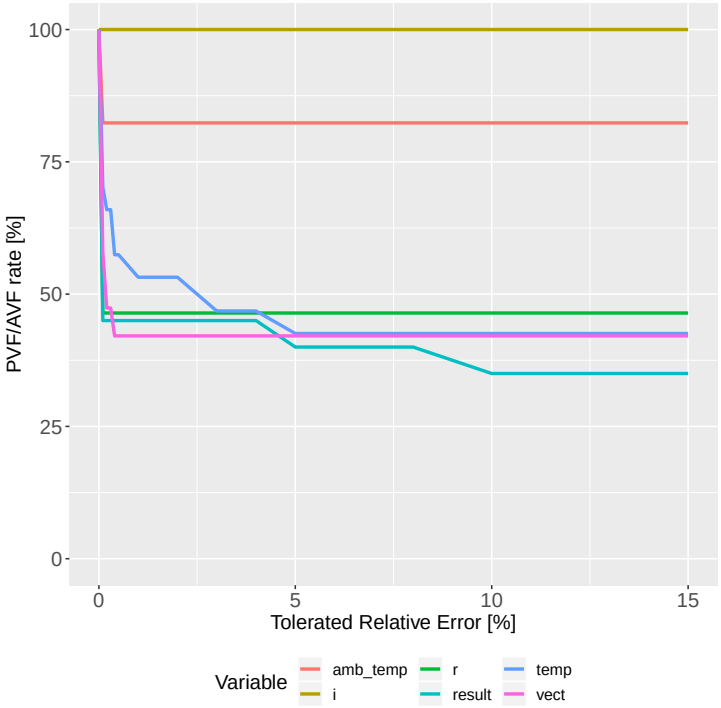
Fonte: Autor

Quadro 3 – Ordenação das variáveis do Hotspot para enrobustecimento seletivo.

Classificação	**Variável**	**PVF** [%]
1ª	i	0.66
2ª	r	1.43
3ª	vect	0.97
4ª	temp	2.41
5ª	amb_temp	0.87
6ª	result	1.59

Fonte: Autor.

Figura 5 – PVF por erro relativo tolerado para o Hotspot.



Fonte: Autor

No gráfico, é possível notar que existem variáveis cujo PVF não muda substancialmente quando o erro relativo máximo permitido na saída é aumentado. Esse é o caso para as variáveis *i* e *amp_temp*. Nesses casos, a variável *i* sempre se mantém com o mesmo PVF, enquanto o PVF da variável *amp_temp* diminui até, no máximo, aproximadamente 82.25%. Em contrapartida, as outras variáveis possuem um comportamento mais interessante, em que todas têm o PVF diminuído por, pelo menos, 50% até um relativo máximo tolerado de 4%.

5.4.4 LavaMD

Na Figura 6, o gráfico do PVF de cada variável em função do erro relativo máximo na saída pode ser observado. Os eixos horizontal e vertical são exatamente como descritos na subseção 5.4.2 e na subseção 5.4.3.

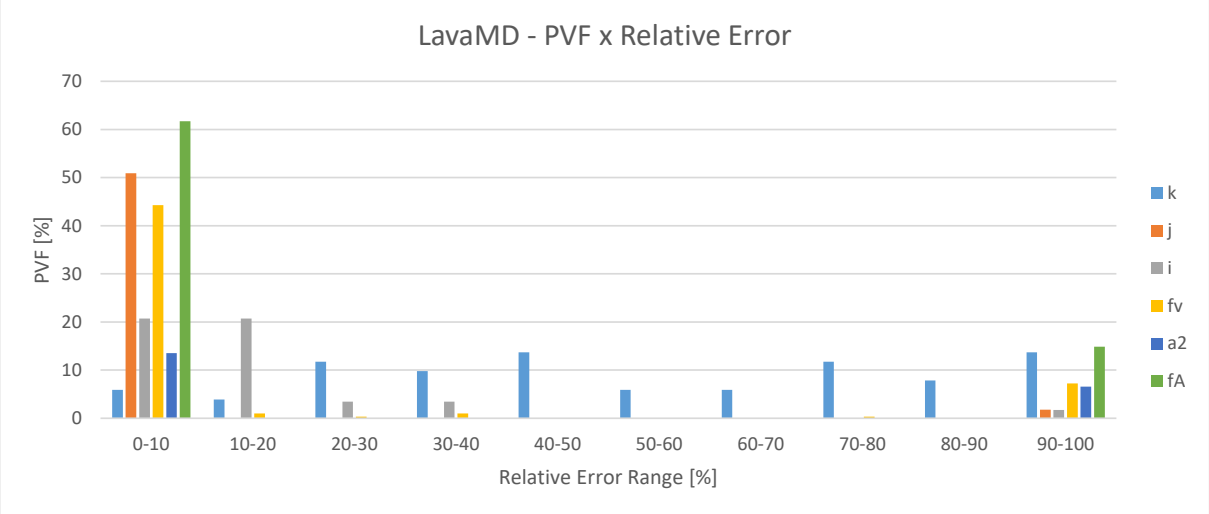
No gráfico, é possível observar alguns comportamentos que são distintos daqueles observados para os outros benchmarks. A primeira característica é que todas as variáveis (com exceção da variável k) possuem um PVF mais elevado na região de 0% até 10% do que nas outras regiões. Isso sugere que, mesmo que o enrobustecimento seletivo seja aplicado, um relaxamento no erro relativo máximo tolerado pela aplicação pode ser uma maneira interessante de diminuir a quantidade de valores incorretos na saída. A segunda característica é que existe uma variável, a dizer, k, que possui valores incorretos que se estendem por todas as regiões, não se limitando a somente algumas delas.

De mão das observações anteriores, podemos realizar uma ordenação das variáveis candidatas para receberem o enrobustecimento seletivo. Essa ordenação pode ser observada no Quadro 4.

Finalmente, na Figura 7, é possível observar o comportamento do PVF de cada variável em função do erro relativo máximo permitido na saída. Os eixos horizontal e vertical são exatamente como definidos na subseção 5.4.2 e na subseção 5.4.3.

Pelo gráfico, é possível notar que o comportamento do LavaMD é substancialmente diferente daquele do DGEMM e do Hotspot no que diz respeito ao comportamento do PVF para o erro relativo máximo tolerado na saída. Nesse caso, todas as variáveis têm uma diminuição no PVF a partir de, no máximo, 5% de erro relativo tolerado na saída.

Figura 6 – PVF por erro relativo para o LavaMD.



Fonte: Autor

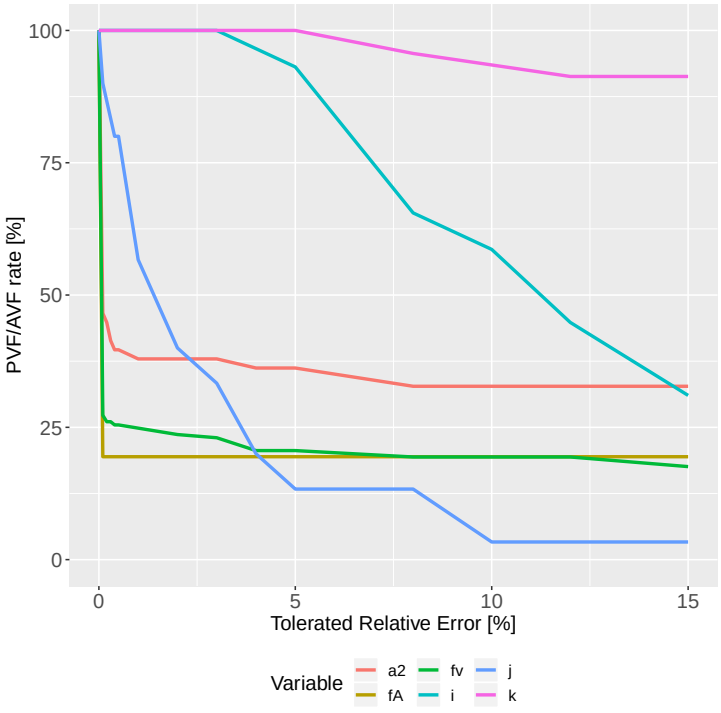
Quadro 4 – Ordenação das variáveis do LavaMD para enrobustecimento seletivo.

Classificação	**Variável**	**PVF** [%]
1ª	fA	1.19
2ª	k	1.52
3ª	fv	5.47
4ª	a2	1.92
5ª	j	0.99
6ª	i	0.96

Fonte: Autor.

Isso contrasta com os outros benchmarks, uma vez que havia variáveis que só mudariam de comportamento a partir de 90% de erro relativo máximo permitido. Além disso, a maioria das variáveis possui uma diminuição significativa do PVF (para até no máximo 37.5%) para um erro relativo máximo tolerado de até 3%. Ainda, a variável j tem o PVF notoriamente diminuído para pouco mais de 5% para um erro relativo máximo tolerado de 10%.

Figura 7 – PVF por erro relativo tolerado para o LavaMD.



Fonte: Autor

6 Ferramenta de automatização

A última contribuição deste trabalho é em relação ao desenvolvimento de uma ferramenta capaz de prover a infraestrutura necessária para a implementação do enrobustecimento seletivo.

A descrição funcional da ferramenta é tratada na seção 6.1. A arquitetura em alto-nível da ferramenta é detalhada na seção 6.2. Os detalhes de implementação da ferramenta são descritos na seção 6.3. Finalmente, o estado final da implementação da ferramenta é descrito na seção 6.4.

6.1 Descrição da ferramenta

A ferramenta desenvolvida neste projeto é denominada *Parsec*. Essa ferramenta inclui um parser da linguagem de programação C11 e pode ser utilizada como infraestrutura para gerar códigos-fonte instrumentados e com enrobustecimento seletivo como saída a partir de um código pré-processado de entrada.

O nome da ferramenta, *Parsec*, foi escolhido a partir de uma modificação na palavra *parser*, de forma que o nome faz alusão à unidade de medida astronômica de mesmo nome. No caso, 1 parsec corresponde a aproximadamente 3.086×10^{13} km.

A ferramenta pode ser utilizada para implementar DMR nas variáveis consideradas sensíveis do código. Em outras palavras, cada variável que recebe o enrobustecimento seletivo é lida e escrita duas vezes, duplicando as operações em cada um dos casos. Caso algum dos resultados nas leituras ou nas escritas sejam diferentes entre si, sabemos que ocorreu uma falha e, então, um relatório de erros é gerado para o usuário.

O objetivo do *Parsec* é permitir a aplicação do enrobustecimento seletivo nas variáveis escolhidas. A possível forma de como essas variáveis podem ser selecionadas é através de uma instrumentação dinâmica de código. Na instrumentação, o usuário seleciona quais variáveis quer avaliar e, então, executa o *Parsec* no modo de instrumentação. Então, um código-fonte na linguagem C11 é gerado e pode ser compilado através de um compilador como, por exemplo, o *gcc*. O código-fonte instrumentado adiciona trechos de código na aplicação original, que obtém informações sobre as variáveis especificadas. Dentre essas informações, estão o número de leituras e de escritas. Neste trabalho, a premissa para que uma variável seja considerada crítica é que ela possua um número significativo de leituras ou de escritas. Nesse caso, se ocorre uma falha que modifica o valor lido ou escrito, é possível que a saída da aplicação seja afetada e que seja produzido um resultado incorreto.

Dessa forma, a utilização do *Parsec* poderá ser feita da seguinte forma: primeiro,

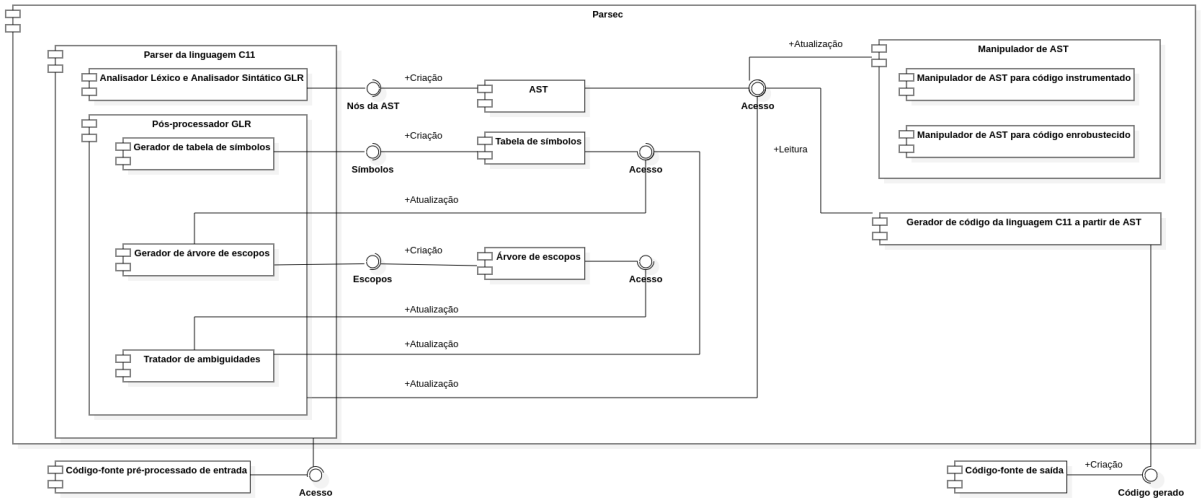
será necessário selecionar as variáveis candidatas para a instrumentação de código; segundo, o código-fonte instrumentado deverá ser compilado e executado, permitindo a obtenção dos relatórios de utilização de cada variável selecionada para a instrumentação; finalmente, as variáveis mais críticas deverão ser adicionadas para o enrobustecimento seletivo.

A implementação da instrumentação dinâmica e do enrobustecimento seletivo estão fora do escopo deste trabalho.

6.2 Arquitetura da ferramenta

A arquitetura do *Parsec* consiste de três grandes blocos: parser da linguagem C11, manipuladores de árvore abstrata de sintaxe (AST, do inglês *Abstract Syntax Tree*) e gerador de código a partir de uma AST. O diagrama de componentes da Figura 8 descreve a arquitetura do *Parsec* em alto nível e é utilizada como referência para a descrição da arquitetura.

Figura 8 – Arquitetura do *Parsec*.



Fonte: Autor

Dessa arquitetura, os manipuladores de AST poderão ser implementados no futuro.

6.2.1 Parser da linguagem C11

A linguagem C11 foi escolhida para ser a principal linguagem reconhecida pela ferramenta. Essa escolha é devida à grande predominância que a linguagem C11 tem sobre outras linguagens de programação no contexto de HPC. As principais motivações para este trabalho vêm justamente sobre o impacto de falhas devidas a radiação ionizante sobre dispositivos paralelos utilizados em HPC e, portanto, a escolha da linguagem C11 é justificada.

De posse da linguagem de programação dos códigos-fonte de entrada, é necessária a utilização de um parser para o reconhecimento do conteúdo desses arquivos. O parser da linguagem C11 implementado neste projeto recebe como entrada um código-fonte escrito em C11 e, então, gera como saída uma AST. A AST é uma representação do código-fonte inicial que desconsidera fatores como pontuação ou outros elementos da linguagem que não possuem caráter funcional. Os dois principais processos envolvidos na execução do parser são a *análise léxica* e a *análise sintática*. A análise léxica é responsável por extrair elementos léxicos importantes do código-fonte. Esses elementos são denominados *tokens*. Dentre os tokens possíveis estão especificadores de tipos (e.g., o texto "int" se torna um token *TK_INT*), identificadores de variáveis (e.g., o texto "int a" se torna *TK_INT* e *TK_IDENTIFIER*, onde o valor textual do último token é "a"), operações aritméticas (e.g., o texto "+" se torna *TK_PLUS*), dentre outros. O analisador sintático, então, recebe a sequência de tokens gerada pelo analisador léxico e extrai os elementos sintáticos importantes. A sintaxe da linguagem a ser reconhecida é descrita através de uma *gramática*. Se o código-fonte de entrada não viola nenhuma das regras especificadas na gramática, a entrada é dita reconhecida. Durante o processo de reconhecimento da linguagem, o analisador sintático gera, para cada construção sintática importante, um ramo de uma árvore. Se a linguagem é reconhecida, o analisador sintático reúne todos os ramos de árvores gerados e cria uma única árvore, que é a AST.

A ciência por trás de parsers é vasta e com certeza não se encontra no escopo deste trabalho. Porém, é importante destacar um ponto relativo a gramáticas: uma gramática pode conter uma *ou mais* regras que acomodem a mesma sequência de tokens gerada pelo analisador léxico. Em outras palavras, gramáticas podem conter *ambiguidades*. Na presença de uma ambiguidade, existem vários métodos que podem ser empregados para lidar com a situação. Uma possibilidade é reestruturar a gramática de forma que a ambiguidade seja eliminada. Outra possibilidade é realizar uma comunicação entre o analisador léxico e o analisador sintático, de forma que tokens diferentes sejam gerados dependendo do histórico dos últimos tokens gerados. A primeira abordagem tem a desvantagem de exigir uma modificação na gramática, o que pode ser complexo e que pode acabar eliminando a clareza das regras. A segunda abordagem não exige, necessariamente, uma modificação nas regras da gramática, mas pode levar a um código difícil de manter e depende diretamente das ferramentas utilizadas para a escrita do analisador léxico e do analisador sintático.

Uma outra abordagem, que é utilizada neste trabalho, é utilizar uma ferramenta que permita a existência de ambiguidades na gramática. Nesses casos, em vez de ser necessário lidar explicitamente com a ambiguidade para, então, continuar com o reconhecimento da linguagem, o parser simplesmente divide a linha do tempo e realiza o reconhecimento de todos os caminhos possíveis. Para cada divisão de linha do tempo, a AST contém um nó marcador que sinaliza a divisão. O resultado final é, efetivamente, uma *floresta* ou uma "árvore de árvores", onde cada sub-árvore corresponde a um caminho completamente

distinto no reconhecimento da entrada. O trabalho de decidir qual caminho é válido ou não é deixado para uma etapa posterior ao reconhecimento da linguagem. Isso evita que sejam necessárias modificações artificiais tanto no analisador léxico quanto no analisador sintático. A desvantagem dessa abordagem reside na performance, pois tanto o reconhecimento da linguagem quanto o pós-processamento da floresta de saída são custosos em termos de quantidade de operações e acessos à memória. No entanto, a vantagem se encontra na facilidade de manutenção de código e na clareza da implementação.

Devido à escolha de permitir o reconhecimento de gramáticas com ambiguidades, é necessária uma etapa de desambiguação da AST, que ocorre após a execução do analisador léxico e do analisador sintático. Para que a desambiguação seja possível, é necessário obter informações como os *símbolos* e os *escopos* presentes no código-fonte. Os símbolos são, em geral, identificadores de funções, de variáveis, e de tipos definidos pelo usuário através da construção *typedef*. O gerador de tabela de símbolos percorre a AST e adiciona cada símbolo a uma tabela em memória. Os escopos são abertos e fechados de acordo com a declaração de funções e com blocos de código. O gerador de árvore de escopos percorre a AST e adiciona todos os escopos presentes em uma árvore em memória. Tanto a tabela de símbolos quanto a árvore de escopos são utilizadas em conjunto para decidir se um caminho na AST é válido ou não. O componente do *Parsec* que realiza essas decisões é o tratador de ambiguidades. A saída do tratador de ambiguidades é uma AST livre de ambiguidades que descreve a estrutura do código-fonte original corretamente.

6.2.2 Manipuladores de AST

Os manipuladores de AST poderão ser compostos por dois componentes: o manipulador de AST para geração de código instrumentado e o manipulador de AST para geração de código com enrobustecimento seletivo. Os manipuladores são executados após o parser e recebem como entrada a AST sem ambiguidades. O procedimento padrão é realizar uma instrumentação de código, obter as informações sobre as variáveis após a execução do código e, então, utilizar essas informações para realizar o enrobustecimento seletivo. Portanto, essas tarefas são mutualmente excludentes.

Os manipuladores percorrem a AST e procuram por declarações de variáveis contendo os especificadores de instrumentação e de enrobustecimento seletivo. O manipulador para instrumentação obtém essas variáveis e gera funções auxiliares para realizar a avaliação em tempo de execução. Essas funções poderão ser adicionadas na AST como se estivessem presentes no código-fonte desde o começo. Da mesma forma, no caso do manipulador para enrobustecimento seletivo, as construções de DMR e de relatório de erros necessárias para o enrobustecimento poderão ser adicionadas na AST como se fossem parte do código-fonte inicial. A saída dos manipulador é, então, uma AST modificada para incluir novos trechos de código.

6.2.3 Gerador de código a partir de AST

Após os manipuladores gerarem as ASTs modificadas, é necessário que os nós das ASTs sejam traduzidos para código da linguagem C11. O componente responsável por essa etapa é o gerador de código.

O gerador de código percorre a AST e, para cada tipo de nó, gera um código C11 associado. A geração da AST pelo parser é tal que existe uma correspondência biunívoca entre um código-fonte da linguagem C11 e uma AST do *Parsec*. Dessa forma, é possível gerar código a partir da AST sem maiores dificuldades. A saída do gerador de código é um código-fonte na linguagem C11. Esse código-fonte pode ser compilado através de qualquer compilador C11 como, por exemplo, o *gcc*.

6.3 Implementação da ferramenta

As linguagens de programação utilizadas para o desenvolvimento do *Parsec* são *C* e *Python*. O analisador léxico e o analisador sintático, bem como o gerador de código a partir de AST são escritos exclusivamente na linguagem C. As etapas de pós-processamento da AST com ambiguidades e manipulação da AST para instrumentação e enrobustecimento seletivo são realizadas primariamente em Python.

6.3.1 Parser da linguagem C11

6.3.1.1 Analisador léxico e analisador sintático

O analisador léxico é implementado através da ferramenta *flex*, versão 2.6.0. O analisador sintático é implementado através da ferramenta *bison*, versão 3.0.4. Ambas as ferramentas trabalham em conjunto, pois o *bison* gera a AST enquanto o *flex* gera a sequência de tokens a partir do código-fonte de entrada.

A maneira pela qual o *flex* obtém os tokens é através de *expressões regulares*. As expressões regulares que descrevem cada token estão contidas no arquivo *c11_lexer.l*. A declaração dos tokens possíveis é feita dentro do arquivo do *bison*: *c11_parser.y*. Esse arquivo contém, além da declaração dos tokens, toda a definição da gramática que define a linguagem C11. Tanto as informações das expressões regulares que definem os tokens quanto da gramática utilizada para o reconhecimento foram obtidas através do *standard* da linguagem C11.

O *bison* é, por padrão, um parser do tipo LALR(1) (*Look-Ahead LR(1)*, onde "LR" significa *left-to-right, rightmost derivation*). Uma explicação detalhada sobre essa definição foge do escopo deste trabalho. No entanto, um fato importante é que um parser desse tipo não suporta gramáticas ambíguas, o que causa os problemas levantados na subseção 6.2.1. Convenientemente, o *bison* também é capaz de gerar um parser do tipo GLR (*Generalized*

LR), que suporta gramáticas ambíguas. Para isso, foi criado um tipo especial de nó na AST chamado de *AST_GLR_SPLIT*. A presença desse nó sinaliza que houve uma divisão da linha do tempo do parser devido a uma ambiguidade na gramática. A AST gerada deve ser, portanto, submetida a um pós-processamento para que todos caminhos inválidos sejam eliminados, produzindo uma AST livre de ambiguidades.

6.3.1.2 Tratador de ambiguidades

A ideia original para a implementação do tratador de ambiguidades era que a linguagem de programação fosse C. No entanto, as operações envolvendo árvores eram de difícil escrita e sofriam constantemente de erros cuja depuração levava um tempo considerável. O principal motivo desses problemas era o baixo-nível da linguagem C, cuja estrutura favorecia a performance, mas dificultava significativamente o desenvolvimento das funcionalidades. Assim, por uma decisão de projeto, o tratador de ambiguidades está sendo atualmente implementado na linguagem Python, que provê diversas facilidades para manipulação de estruturas de dados, em particular, árvores. A AST gerada pelo *bison* é salva em arquivos que são lidos pelo tratador de ambiguidades implementado em Python. A partir desse momento, um script em Python é responsável por realizar todas as manipulações necessárias na AST e, então, retornar a execução para o programa em C.

Um dos principais motivos para a escolha de um parser GLR em vez de um parser LALR(1) foi a presença da construção *typedef* do C11, que permite a definição de tipos pelo usuário. A existência dessa construção é uma fonte de inúmeros problemas relativos a ambiguidades na gramática do C11. De fato, a presença de typedefs no código-fonte provoca uma quantidade significativa de divisões na linha do tempo do parser GLR. Portanto, uma das principais tarefas do tratador de ambiguidades é realizar a eliminação de caminhos inválidos gerados por typedefs no código-fonte. Por exemplo, considere o caso onde não se sabe se um identificador “foo”, que aparece na construção “int foo = 0;”, é um tipo definido pelo usuário ou um identificador de variável. Nesse caso, o tratador de ambiguidades verifica os dois caminhos possíveis do nó *AST_GLR_SPLIT* na AST. Se um dos caminhos considera que “foo” é um especificador de tipo, então esse é um caminho inválido, uma vez que apenas “int” é um especificador, e não “foo”. Assim, o caminho inválido é removido da AST, o nó *AST_GLR_SPLIT* é excluído e o nó pai do caminho válido assume a posição que o nó *AST_GLR_SPLIT* assumia antes de ser excluído. Agora, o identificador “foo” é corretamente reconhecido como um identificador de variável, e não um tipo definido pelo usuário.

Alguns cuidados devem ser tomados quando existem construções mais complexas como *structs* e comandos de controle de fluxo, como *if* e *else*. Algumas ambiguidades podem surgir nesses contextos. Porém, em geral, as ambiguidades são de fácil eliminação. Por esse motivo, a etapa de eliminação de ambiguidades devidas a typedefs é chamada de

fase grossa, enquanto a etapa de eliminação dos outros tipos de ambiguidades é chamada de *fase fina*. A divisão do tratador de ambiguidades nessas duas etapas torna as tarefas menos dependentes e o código mais fácil de manter.

6.3.2 Manipuladores de AST

Os manipuladores de AST serão os componentes que implementam os trechos de códigos instrumentados e os trechos de códigos com enrobustecimento seletivo.

Para a instrumentação, o *Parsec* poderá percorrer a AST procurando por variáveis em que algum dos especificadores seja o especificador de instrumentação. Então, para cada leitura e escrita dessas variáveis, o *Parsec* gera nós adicionais na AST, que correspondem aos trechos da instrumentação. Cada função de leitura e de escrita mantém informações sobre a variável em questão como, por exemplo, o número de leituras e de escritas. Essas informações serão armazenadas em variáveis globais e atualizadas por funções de instrumentação.

Para o enrobustecimento seletivo, a abordagem será semelhante. A diferença é que as variáveis marcadas com o especificador de enrobustecimento seletivo serão duplamente verificadas em operações de leitura e de escrita como detalhado na seção 6.1.

6.3.3 Gerador de código a partir de AST

A tradução de uma AST para um código-fonte válido da linguagem C11 é realizada pelo gerador de código.

Como a construção da AST é tal que existe uma correspondência biunívoca entre um código-fonte da linguagem C11 e os nós da AST, o gerador de código é, efetivamente, o inverso do processo que o parser da subseção 6.3.1 realiza. O gerador de código percorre toda a AST e, para cada nó, gera um texto que corresponde exatamente à estrutura da regra correspondente na gramática utilizada pelo *bison*. Assim, todos os nós da AST são convertidos para texto. A saída desse processo é um código-fonte válido na linguagem C11, que reflete precisamente o código-fonte original e, adicionalmente, os trechos de códigos instrumentados e de códigos com enrobustecimento seletivo.

6.4 Estado final da implementação da ferramenta

A ferramenta, nesse estágio do projeto, é capaz de ler códigos-fonte na linguagem C11 corretamente, realizar a desambiguação da floresta gerada e, finalmente, também é capaz de gerar um código-fonte válido a partir de uma AST sem ambiguidades. Portanto, a ferramenta provê toda a infraestrutura necessária para a implementação da instrumentação dinâmica e do enrobustecimento seletivo.

7 Conclusão

Este trabalho provê uma análise da criticidade dos erros na saída de aplicações paralelas causados por falhas devidas à radiação ionizante. Análise conduzida estabelece uma lista de variáveis ordenada pela criticidade de cada variável. Os métodos desenvolvidos para realizar essa análise podem ser aplicados a qualquer aplicação paralela que possa ser executada em um ambiente de injeção de falhas como o CAROL-FI.

A análise de criticidade mostrou que a distribuição dos erros nas variáveis da saída das aplicações pode variar significativamente dependendo de qual aplicação é considerada. Para cada uma das 3 aplicações consideradas, cada distribuição possuiu tendências diferentes que, em última análise, implicaram em uma classificação diferente para a aplicação do enrobustecimento seletivo.

Ainda, as análises sobre o comportamento do PVF de cada variável em função do erro relativo máximo permitido na saída confirmou que, para certos casos, um relaxamento na definição de corretude dos valores de saída pode reduzir significativamente a quantidade de dados incorretos observados.

A implementação de uma ferramenta que pode ser utilizada para realizar o enrobustecimento seletivo poderá ser utilizada em trabalhos futuros para confirmar a validade das previsões teóricas deste e de outros trabalhos já realizados sobre o assunto. Com o que foi desenvolvido até então, mais aplicações poderão ser consideradas em uma análise semelhante e, com certeza, no futuro será possível aplicar a instrumentação dinâmica e a duplicação automaticamente através da ferramenta de automatização da técnica de enrobustecimento seletivo.

Finalmente, o autor espera que os resultados levantados neste trabalho possam ser utilizados em diferentes aplicações da Ciência e, em particular, da Engenharia e da Computação. Devido à generalidade das análises aqui apresentadas, o autor espera que esse objetivo possa ser atingido em um futuro próximo e com sucesso.

Referências

BAUMANN, R. C. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability*, v. 5, n. 3, p. 305–316, Sep. 2005. ISSN 1530-4388. Citado 3 vezes nas páginas 14, 19 e 20.

GEIST, A. *How To Kill A Supercomputer: Dirty Power, Cosmic Rays, and Bad Solder*. [S.l.]: IEEE Spectrum, 2016. <http://spectrum.ieee.org/computing/hardware/how-to-kill-a-supercomputer-dirty-power-cosmic-rays-and-bad-solder>. Citado na página 20.

INTEL. *Intel® Core™ i5-3337U Processor*. [S.l.]: Intel, 2019. <https://ark.intel.com/content/www/us/en/ark/products/72055/intel-core-i5-3337u-processor-3m-cache-up-to-2-70-ghz.html>. Citado na página 23.

JEDEC. *Measurement and Reporting of Alpha Particle and Terrestrial Cosmic Ray-Induced Soft Errors in Semiconductor Devices*. [S.l.], 2006. Citado na página 19.

MAHATME, N. N. et al. Comparison of combinational and sequential error rates for a deep submicron process. *IEEE Transactions on Nuclear Science*, v. 58, n. 6, p. 2719–2725, Dec 2011. ISSN 0018-9499. Citado na página 20.

NETTO, V. F. *Parsec repository*. [S.l.]: Vinícius Fratin Netto, 2019. <https://bitbucket.org/viniciusfratin/parsec/>. Citado na página 17.

OLIVEIRA, D. et al. Carol-fi: An efficient fault-injection tool for vulnerability evaluation of modern hpc parallel accelerators. In: *Proceedings of the Computing Frontiers Conference*. New York, NY, USA: ACM, 2017. (CF'17), p. 295–298. ISBN 978-1-4503-4487-6. Disponível em: <http://doi.acm.org/10.1145/3075564.3075598>. Citado na página 21.

Oliveira, D.; Navaux, P.; Rech, P. Increasing the efficiency and efficacy of selective-hardening for parallel applications. In: *2019 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. [S.l.: s.n.], 2019. p. 1–6. ISSN 1550-5774. Citado 2 vezes nas páginas 25 e 26.

OLIVEIRA, D. A. G. d. *Hardening strategies for HPC applications*. Tese (Doutorado) — Universidade Federal do Rio Grande do Sul (UFRGS), 2017. Citado 2 vezes nas páginas 21 e 22.

OLIVEIRA, D. A. G. de et al. Evaluation and mitigation of radiation-induced soft errors in graphics processing units. *IEEE Transactions on Computers*, v. 65, n. 3, p. 791–804, March 2016. ISSN 0018-9340. Citado na página 21.

RECH, P. et al. Impact of GPUs Parallelism Management on Safety-Critical and HPC Applications Reliability. In: *IEEE International Conference on Dependable Systems and Networks (DSN 2014)*. Atlanta, USA: [s.n.], 2014. Citado na página 20.

SHEIKH, A. T. et al. A fault tolerance technique for combinational circuits based on selective-transistor redundancy. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, v. 25, n. 1, p. 224–237, Jan 2017. ISSN 1063-8210. Citado na página 20.

SILVA, F. et al. An efficient, low-cost ecc approach for critical-application memories. In: *2017 30th Symposium on Integrated Circuits and Systems Design (SBCCI)*. [S.l.: s.n.], 2017. p. 198–203. Citado na página 20.

TIWARI, D. et al. Understanding gpu errors on large-scale hpc systems and the implications for system design and operation. In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. [S.l.: s.n.], 2015. p. 331–342. ISSN 1530-0897. Citado na página 20.

TOP500. *TOP 10 Sites for June 2016*. [S.l.]: TOP500, 2016. <https://www.top500.org/lists/2016/06/>. Citado na página 20.